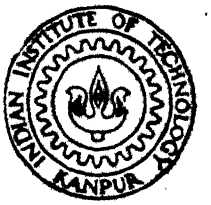
A TEMPORAL LOGIC PROGRAMMING SYSTEM

by

K. C. MUKHERJEE



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988

A TEMPORAL LOGIC PROGRAMMING SYSTEM

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

K. C. MUKHERJEE

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1988

CSE-1988-M-MUK-TEM

CERTIFICATE

This is to certify that the thesis entitled **A TEMPORAL LOGIC PROGRAMMING SYSTEM** is a report of the work carried out under our supervision by **Krishna Charan Mukherjee** , and that it has not been submitted elsewhere for a degree.

Dr. H. Karnick
Asst. Professor
Department of
Computer Science
and Engineering
I.I.T. Kanpur

Dr. R. Sangal
Asst. Professor
Department of
Computer Science
and Engineering
I.I.T. Kanpur

Place : Kanpur , India

Date : February , 1988.

ACKNOWLEDGEMENT

It gives me great pleasure to put on record my feeling of gratitude to Dr. R. Sangal and Dr. H. Karnick, my thesis supervisors. They suggested the problem and created an atmosphere of freedom in which it was a pleasure to work. In spite of their busy schedules, they could spare enough time to clarify my doubts and ensure smooth progress.

Special words of gratitude are due to my classmates (M.Tech, CS 1986-87) and residents of Hall 4 without whom I.I.T-K would have been 18 months of boredom.

ABSTRACT

The aim of the thesis is to explore planning and understanding. Planning involves selecting a sequence of actions to reach a desired state. It includes assessing a situation, deciding what goals to pursue, creating plans to secure these goals and executing plans. Understanding concerns the way in which a situation is comprehended. In this work, we have emphasised on temporal understanding of a situation. We have dealt with the issue of reasoning about time.

The logic programming formalism has been chosen for the purpose of developing a plan generator and a temporal system analyser. In this formalism, the user programs (in the form of rules of a logic program) are easier to read. They are not cluttered up with details of how things are to be done - they will be more like specifications of what a solution will look like.

The design of a temporal logic programming system helps in overcoming the problem of specification of frame axioms which are otherwise needed for handling the notions of time, state and history by means of a logic program. In addition, the designed system offers a powerful query mechanism for reasoning about temporal relationships.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Planning and Understanding

How do people understand natural language ? How do they behave rationally in a variety of situations ? How can computers be made to do likewise ?

The researcher in the field of Artificial Intelligence is interested in extending the capabilities of the computer system to encompass common-sense reasoning capabilities. He feels the need to explore the nature of knowledge : how it can be represented, how it can be stored and accessed and how it can be used in various tasks thought to constitute cognition.

The aim of the thesis is to explore such possibilities about two areas of cognition - planning and understanding. In R. Wilensky's [14] words "Planning concerns the process by which people select a course of action - deciding what they want, formulating and revising plans, dealing with problems and adversity, making choices and eventually performing some action ... planning includes assessing a situation, deciding what goals to pursue, creating plans to secure these goals and executiong plans. Understanding concerns the way in which a person comprehends a situation -

inferring implicit components, establishing coherence of an episode, structuring events into meaningful units and finding explanations for various actions." Thus understanding involves reasoning about plans and action of the people in a situation. In this work, we have emphasised on temporal understanding of a situation. We have dealt with the issue of reasoning about time.

The popular logic programming formalism has been chosen for the purpose of developing a plan generator and a temporal system analyser. This allows reasoning to be done in a neat way. The user programs (in the form of rules of a logic program) are easier to read. They are not cluttered up with details of how things are to be done - they will be more like specifications of what a solution will look like. The design of a logic programming shell for the purpose of plan generation and temporal analysis helped in circumventing the problem of specification of frame axioms which are otherwise needed for handling the notions of time, state and history.

1.2 Overview of the thesis

The rest of the thesis describes the design of a temporal logic programming formalism and its implementation. In chapter 2, the deficiencies of the existing system for planning in a logic programming environment are highlighted. This has motivated the need for developing a new formalism which will assist the user in planning and reasoning about

time in a more natural way. In chapter 3, the development of a temporal logic programming formalism is outlined. In chapter 4, we describe a logic programming system built by using the new formalism. Chapter 5 is an analysis of the designed system. In this chapter, we highlight the efficiencies of the system and also suggest areas of improvement. The appendices at the end provide the set of rules for the example domain - Blocks World. We also present an interactive session with the temporal logic programming system.

CHAPTER 2

HANDLING TIME, STATE AND HISTORY IN LOGIC PROGRAMMING

2.1 Introduction

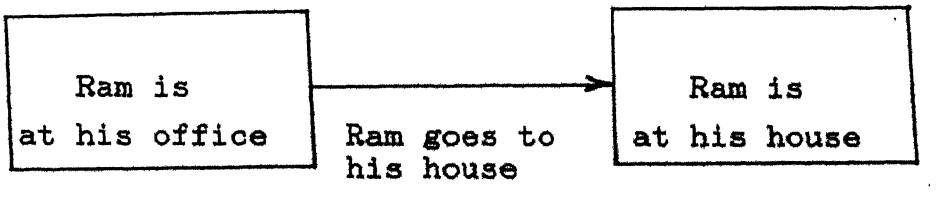
The work described here arose in trying to use the logic programming formalism for question answering in the context of story understanding. The idea is that since answering questions is based on causal connections between events and states, it should be possible to use the logic programming formalism fairly naturally. However, this raises the issues of how time, state and history can be handled in logic programming. The state can be incorporated as an additional argument of predicates. This will involve rewriting of the existing rules of the logic program to take care of the state explicitly. The other alternative is to modify the logic programming shell so that it handles state, time and history implicitly as it executes a logic program. The user is expected to broadly guide the system. The problem of handling state explicitly is obviated in the latter approach.

2.2 States and Events

A *state* is something that is true for a while; false

for a while, and so on, like the state of Ram being at his house or elsewhere. An *event* is something that can happen like Ram going to his house.

Events are transitions between states. A state token is true over an interval of time whereas an event *happens* ; it is either instantaneous or has a beginning, middle and end. Practical programs have to worry about these concepts for two reasons. First, things change. A realistic database must keep track of what is true at various times and not just the present. Second, the machine may have at its disposal ways of intervening in the world in order to achieve its goals.



2.3 An Example Domain - The Blocks World

The Blocks World [11,13] is chosen to illustrate the concept of states, actions and how states change due to actions. The world consists of a plane surface or table with blocks on it. The blocks are labelled and may be stacked in various ways. The position and configuration of the blocks are known. There is a robot arm that can move the blocks.

Knowledge about the Blocks World is specified in terms of general rules, and inferring the solution to a problem is done by instantiating the rules to a particular situation.

Knowledge is represented as Horn clauses in a PROLOG [3] like system. Knowledge will be represented as *rules* and *facts*. The *rules* are applicable to a variety of situations. They usually contain variables. *Facts* are specific and do not contain variables.The following is a fact :- *(on A B)* which states that block *A* is on top of block *B* . It consists of a predicate *on* applied to *A* and *B* . The predicate *on* is true of two blocks if and only if the first one is on top of the second. A predicate applied to arguments is called predication. The rules are implications consisting of a left hand side and a right hand side. The left hand side has a predication and the right hand side has one or more predications. Its meaning is that the left hand side can be inferred provided the predications on the right hand side hold. In a rule , the prefix *?* will be reserved for logical variables.These variables may vary over any object in the database. The following is an example of a rule :-

(over ?x ?y) <- (block ?x) (block ?y) (on ?x ?y) .

It asserts that an arbitrary block *?x* is *over* another arbitrary block *?y* if *?x* is on *?y* . *?x* and *?y* are variables that may be instantiated by blocks.

Inference is done in the standard logic programming fashion.To know whether *A* is over *C* , we pose a query like *? (over A C).* The inference proceeds in the

following manner . First, we check whether a query or a goal already matches with a fact already present. If yes, the query is true. In case the query fails to match with any of the given facts, we try matching it with the left hand side (consequent) of the rules. If a rule matches, the predicates on its right hand side (antecedents) after proper instantiation become the subgoals. In case of failure to match a subgoal we try another rule. The process repeats until we are successful or no more rules for a particular predicate remain.

The *Robot Arm* is an important component of the Blocks World problem. It can perform three operations. It can open open or close its gripper to grasp or ungrasp a block. It can also be made to move to a desired position. The three special predicates that relate to the actions of the robot arm are :-

(grip) -- close the gripper

(ungrip) -- open the gripper

(move-robot-arm ?x ?y ?z) -- move the robot arm to ?x, ?y & ?z coordinates of a three-dimensional space

Determining the truth value of these predicates causes the robot arm to be actuated. For modelling the state of the Blocks World , the predicates are as follows:-

(posn ?block ?x ?y ?z) -- true in the state where the position of the ?block is ?x , ?y and ?z.

(on ?block1 ?block2) -- true in the state where ?block1 is on top of ?block2.

(clear ?block) -- true in the state where ?block is clear (i.e. no other block is on it).

(grasp ?block) -- true in the state where ?block is clear and gripped by the robot arm.

(posn-hand ?x ?y ?z) -- true in the state where the robot arm is positioned at coordinates ?x , ?y and ?z.

(handempty) -- true in the state where the robot arm is not gripping any block and is empty.

The truth values of these predicates describe a particular state. When the state of the Blocks World changes i.e. we move from one state to another there is a change in the truth value of one or more of these state description predicates.

2.4 State as an argument of predicates

With the idea of keeping track of the changing database of facts with time as well as to answer time-related questions, the first solution attempted was by incorporating a *state* or *situation* variable in each of the predicates whose truth values can change from state to state. The idea was in accordance with Green's formulation [5,1Ø]. By this, the robot problems are formulated in such a way that a resolution theorem proving system can solve them. This formulation involved one set of assertions that

described the initial state and another set that described the effects of various robot actions on the states. The goal condition was then described by a formula with an existentially quantified state variable. That is, the system would attempt to prove that there existed a state in which a certain condition was true. A constructive proof method could then be used to produce the set of actions required to create the desired state. In other words, the entire *plan* or the sequence of primitive actions needed for moving from the initial state to the final state would be unified with the state variable of the goal formula. The deduction system used was the logic programming shell - "VIDHI" [12]. An example to illustrate the above notion is as follows :-

Suppose the primitive actions which constitute a plan are

(1) *(grasp ?block)*

(2) *(ungrasp ?block)* and

(3) *(move-hand-to ?x ?y ?z)*

where the symbols have their usual significance. Let the initial state be designated by *SØ*. An example configuration of blocks in the initial state is given by the following facts in VIDHI :-

```
(defasrt 1 (initial-state SØ))

(defasrt 2 (block A))

(defasrt 3 (block B))

(defasrt 4 (is-posn A 1 1 1 SØ))
```

```
(defasrt 5 (is-posn B 2 2 2 S0))

(defasrt 6 (ht A 2))

(defasrt 7 (ht B 2))

(defasrt 8 (is-clear A S0))

(defasrt 9 (is-clear B S0))

(defasrt 10 (is-posn-hand 3 3 3 S0))
```

From the above the facts that are true about the blocks (*A* and *B*) as well as about the robot-arm are clear . In this method of using state variables explicitly as arguments of predicates, the special functional expression *(*do <action> <state>)* is used to denote the function that maps <state> into another state resulting from the <action> in <state>. Thus by first moving the robot arm to the position (1 1 1) , we move from the initial state S0 to the state given by *(*do (move-hand-to 1 1 1) S0)* . Now , the rule for putting a block on top of another is formulated as :-

```
(defasrt 11 (on ?block1 ?block2
                (*do (ungrasp ?block1)
                (*do (move-hand-to ?x2 ?y2 ?z-comp)
                (*do (grasp ?block1)
                (*do (move-hand-to ?x1 ?y1 ?z1) ?s))))) <-
                (is-clear ?block1 ?s)
                (is-clear ?block2 ?s)
                (is-posn ?block1 ?x1 ?y1 ?z1 ?s)
```

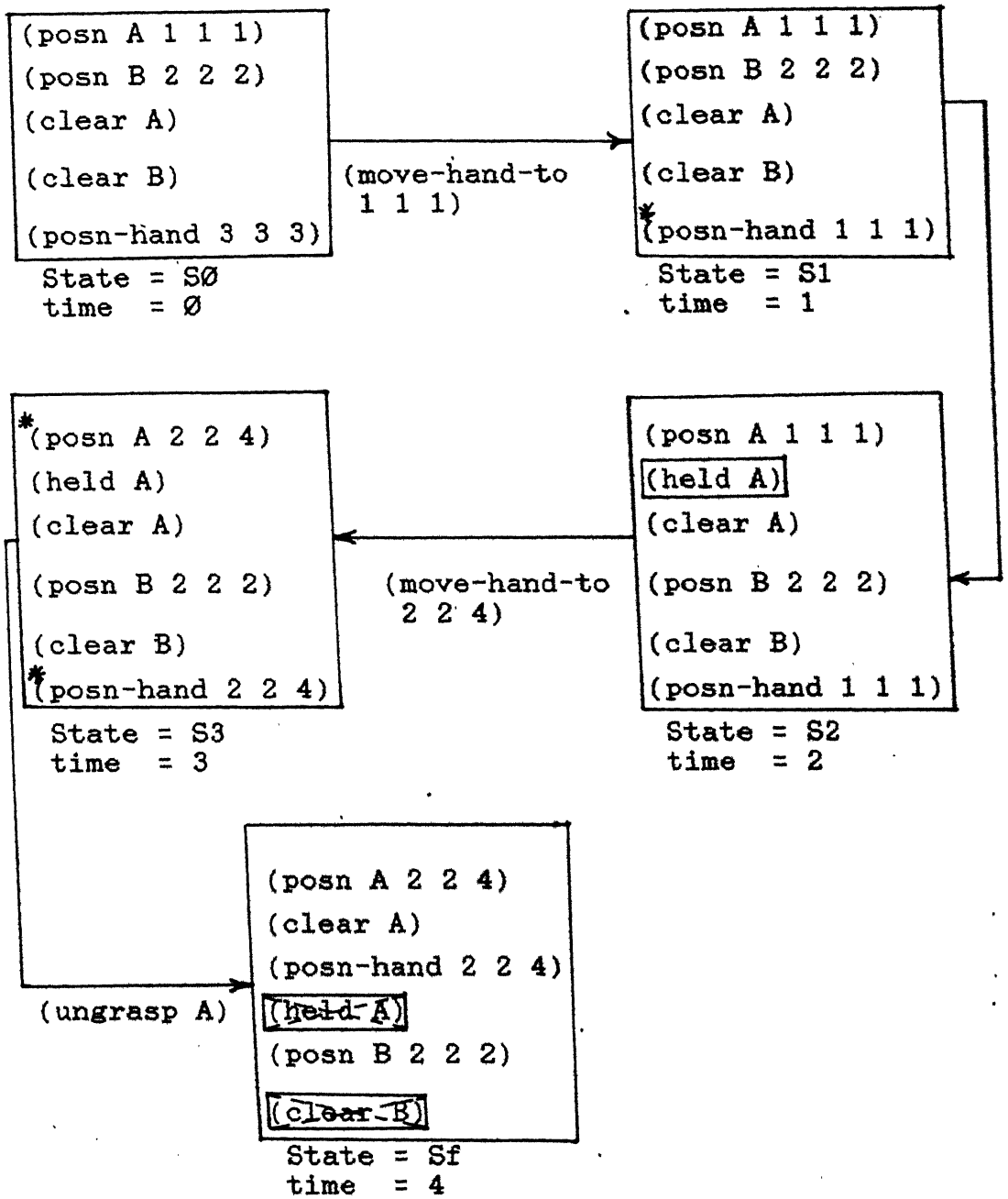
```
(is-posn-hand ?xh ?yh ?zh ?s)
(not (= ?x1 ?xh))
(not (= ?y1 ?yh))
(not (= ?z1 ?zh))
(is-posn ?block2 ?x2 ?y2 ?z2 ?s)
(ht ?block1 ?h1)
(ht ?block2 ?h2)
(= ?z-comp (+ ?z2 (* 0.5 ?h2) (* 0.5 ?h1)))
```

So , when we set a goal as *(goal (on A B ?sf))* , the logic programming formalism is used to derive the answer for the state *?sf* in which *A* is on top of *B* . By using the rule given for *on* and the facts asserted about the initial state *S0* in the database, the value for *?sf* is obtained as

```
?sf = (*do (ungrasp A)
       (*do (move-hand-to 2 2 4)
       (*do (grasp A)
       (*do (move-hand-to 1 1 1) S0))))
```

In this way, the final state *?sf* is a result of executing a series of primitive actions on the initial state *S0*, where each primitive action causes a transition from one state to another. By this method, the *plan* needed for putting the block *A* on top of block *B* is essentially captured in the unifier for the final state variable *?sf*.

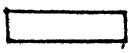
We then explored the possibilities when the state variable is included as an explicit argument of the predicates. Temporal understanding of plans and qualitative reasoning about time were provided as additional features by means of rules of a logic program. The primitive actions, which have been listed before were considered to be *time-setters*, that is, they caused a change in state and also incremented the clock. So, the initial set of facts asserted about the Blocks World are a true description of the initial state at the zeroth instant of time. Subsequently, as a plan is generated to achieve a desired state, in which a certain relation becomes true, the final state is expressed as an effect of the plan on the initial state. In order to proceed from the initial state to the final state by sequentially executing the primitive actions of the plan, we move through several instants of time. Analysing the example mentioned,

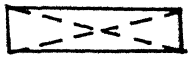


As a transition takes place from one state to another, the following will happen :

(1) Time is incremented since a primitive action has taken place .

(2) Some of the existing state description predicates may change (illustrated by a * in the diagram)

(3) New state description predicates may become true (illustrated by ).

(4) Old state description predicates may cease to be valid in the new state (illustrated by  in the diagram).

This changing state description causes different relations to become true at different instants of time. For example, the relations *(clear A)* and *(clear B)* are coincident in time i.e. they are jointly true at time instants = Ø,1,2 and 3. Again the relation *(posn-hand 3 3 3)* *precedes* the relation *(held A)* in time as the former is true at the time instant = Ø while the latter becomes true at time instant = 2. We could also say that the relation *(held A)* *follows* the relation *(posn-hand 3 3 3)* in time.

When the state is used as an explicit argument of predicates, temporal analysis of the Blocks World for a given plan can be done by the use of additional rules i.e. rules in addition to those required for plan generation. Time comparisons between relations can be made by examining the final state (expressed in terms of the plan and initial state) and the intermediate states.

2.5 The Frame Problem

To use a familiar analogy, the changes between one state description and another can be compared to changes

between frames in an animated film. The problem of specifying which formulas in a state description should change and which should not is called the *frame problem* in Artificial Intelligence. When the state variable is used as an additional argument of predicates and the logic programming formalism is being used to solve the problem of plan generation and temporal analysis we must completely specify the effects of an action on an existing state. We must also state that certain relations are unaffected by the action. Here, the "effects" and "non-effects" alike need to be stated explicitly. Using Green's formulation, we must have assertion for each relation not affected by an action or a sequence of actions. Using a state variable as an explicit argument of predicates, we observe that a number of assertions have to be made to emphasise the "non-effects" of various actions. For example, the following assertion is needed to state the fact that when the robot arm is moved to the position of a block, the block grasped, the hand moved to a different position and then the block released, the position of another block remains unchanged.

```
(defasrt ipb2 (is-posn ?block ?x ?y ?z
                (*do (ungrasp ?block1)
                (*do (move-hand-to ?x2 ?y2 ?z2)
                (*do (grasp ?block1)
                (*do (move-hand-to ?x1 ?y1 ?z1) ?s)))))) <-
                (diffrnt-blocks ?block1 ?block)
                (is-posn ?block ?x ?y ?z ?s))
```

Similarly, by the second assertion, a block remains on top of another when a third different block undergoes the above sequence of actions.

```
(defasrt ion2 (is-on ?block1 ?block
                (*do (ungrasp ?block2)
                (*do (move-hand-to ?x1 ?y1 ?z1)
                (*do (grasp ?block2)
                (*do (move-hand-to ?x ?y ?z) ?s))))) <-
                (diffrnt-blocks ?block ?block1)
                (diffrnt-blocks ?block ?block2)
                (diffrnt-blocks ?block1 ?block2)
                (is-on ?block1 ?block2))
```

By the third assertion, a block continues to be held in the resulting state when the robot arm is moved in a particular state.

```
(defasrt ihb3 (is-held ?block (*do (move-hand-to ?x ?y
?z) ?s)) <-
                (is-held ?block ?s))
```

The predicate *(diffrnt-blocks ?x ?y)* is true when *?x* and *?y* are unified to different blocks.

The assertion describing what stays the same during an action are called the *frame assertions* or *frame axioms*. In large systems, there may be many predicates used to

describe a situation. By using Green's formulation, a separate frame assertion is needed for each predicate. By using Kowalski's formulation [6,7], the statement of the frame assertion may be simplified but not removed totally.

What would ordinarily be predicates in Green's formulation are made terms in Kowalski's formulation. This simplifies the *frame assertions*. Here, only one *frame axiom* is needed for each action. For example, instead of using the literal *(on A D SØ)* to denote the fact that *A* is on *D* in *SØ*, we use the literal *(holds (on A D) SØ)*. The term *(on A D)* denotes the "concept" of *A* being on *D*. Suppose the robot has an action that can "transfer" a block *?x* from position *?y* to position *?z* where *?y* and *?z* might be either names of other blocks that block *?x* might be resting on or names of positions on the table. Let us assume that both block *?x* and position *?z* must be clear to execute the action. We model this by *(trans ?x ?y ?z)*. Now we express part of the effects of the actions by using a separate *holds* literal for each relation made true by the action. For example,

*(holds (clear ?x) (*do (trans ?x ?y ?z) ?s))*

*(holds (clear ?y) (*do (trans ?x ?y ?z) ?s))*

*(holds (on ?x ?z) (*do (trans ?x ?y ?z) ?s))*.

The major advantage of Kowalski's formulation is that we need only one *frame assertion* for each action. In our

example, the single *frame assertion* is :

{ (holds ?v ?s) ∧(diff ?v (clear ?z)) ∧(diff ?v (on ?x ?y)) } ->

*(holds ?v (*do (trans ?x ?y ?z) ?s))*

This expression quite simply states that all terms different than *(clear ?z)* and *(on ?x ?y)* still *hold* in all states produced by performing the action *(trans ?x ?y ?z)* .

2.6 Conclusion

From the preceeding discussion, we see that although plan generation and temporal analysis can be done through the existing logic programming formalism (VIDHI) by incorporation of state as an additional argument of predicates, the process is cumbersome and not attractive to a user of the system. The responsibility of reasoning about states, maintaining the state transitions with actions etc. is forced upon the writer of the logic program. He has to also make a number of *frame assertions* to get an accurate picture of the changing states with time. Further, an equally important disadvantage is the inefficiency of the *frame axioms.*

In order to relieve the user of such burdensome programming, a more sophisticated reasoning system was developed on top of the existing logic programming shell - Vidhi. This enables the experienced user to generate plans and do temporal analysis in a more natural way.

CHAPTER 3

DESIGN OF A TEMPORAL LOGIC PROGRAMMING FORMALISM

3.1 Introduction

In light of the problems mentioned in the previous chapter, we need a logic programming system which would accept user's program (set of rules) and implicitly reason about state, time and history. The user is then relieved of the burden of taking care of state explicitly through his logic program.

In this chapter we describe our temporal logic programming formalism.

3.2 The Model of Planning

3.2.1 Inference as Planning

The desired sequence of actions needed for achieving a goal is called a *plan* for achieving the goal. The goal may be looked upon as a *desired state* into which the planning world is to be transformed. Given the goal, the planner must assess (or observe) the existing situation. If it finds that the current state of the world is not the desired (or the goal) state, it generates the *plan* required for transforming the existing state to the goal state. Thus planning accounts for a number of cognitive processes. They

include assessing a situation, deciding what goals to pursue, creating plans to secure these goals and executing plans. In the context of logic programming, the process of inference of a goal state from an initial state generates the plan as an answer substitution for the state variable when it is used explicitly (see chapter 2). When the state variable is removed the interpreter maintains a history of states from which the plan is generated.

3.2.2 Classification of Predicates

When the above structure of planning is to be captured by means of rules of a logic program states as well as changes in the states are modelled by predicates. Thus the classification of predicates becomes necessary. Given an initial state of the world, the goal would desire to transform the world to some other state and ask for the plan needed for doing so. Referring to the earlier example of the Blocks World, the initial state is *SØ* . In this state, blocks *A* and *B* are lying on the table. Now, a plan may be required to be generated for stacking *A* on *B* . After executing the plan the Blocks World will be transferred to a new state where *A* will be on *B* . The required plan can be generated by setting up the goal *(on A B)* . Here *on* can be looked upon as a *desired state* predicate . A *desired state* predication essentially expresses what should be true in a certain state. Therefore, it first observes whether the predication is true in the existing state or

not. If it is, then nothing has to be done, else actions are to be executed in a certain sequence to transform the world to the desired state where the predication becomes true. This necessitates further classification of predicates into *state observation* and *action* predicates respectively. Summarising the above information, we get

(1) *State Observation* predicates :- These predicates refer to the state implicitly . They do not cause any change in state when tested for satisfaction. In the rules

(posn ?block ?x ?y ?z) (- (is-posn ?block ?x ?y ?z)

(clear ?block) (- (is-clear ?block)

(on ?block1 ?block2) (- (is-on ?block1 ?block2)

is-clear , is-posn , is-on are examples of *state observation* predicates .

In general , the user will specify certain dependency rules among the various *state observation* predicates used in modelling the planning world . Some of these predicates are <u>independent</u> i.e. they do not have any rules defining them in terms of other predicates. The others are <u>dependent</u> on other *state observation* predicates i.e. they have rules where other *state observation* predicates appear as antecedents. In our Blocks World, the different *state observation* predicates are *is-place , is-posn , is-gripped , is-on , is-clear , is-grasp , is-loaded , is-*

handempty , *is-posn-hand* etc. Predicates such as *is-posn* , *is-posn-hand* and *is-gripped* have no rules associated with them and hence are *independent state observation* predicates. The others have rules associated with them - for example,

```
(is-grasp ?block) <- (is-posn ?block ?x ?y ?z)
                     (is-posn-hand ?x ?y ?z)
                     (is-gripped)
```

(2) *Desired State* predicates :- These predicates may cause change in state when tested for truth value. If the desired state does not hold, it may cause actions to be initiated to achieve the desired state. In the above rules, predicates like *on* , *clear* , *posn* are examples of such predicates .

There exists a correspondence between *desired state* predicates and *state observation* predicates. The first rule for any *desired state* predicate involves state observation. It typically has the *desired state* predicate in the consequent and a *state observation* predicate in the antecedent. Only in case of failure are the subsequent action rules executed .

Inferring a *desired state* predicate results in the change in truth value of some *state observation* predicate (the *state observation* may have to be asserted or retracted i.e. its negation may have to be

asserted). For specifying such assertions, a special predicate - *asserth* may be included as an antecedent of the *desired state* predicate. This predicate, in conjunction with a special form of cut facility (see later) can cause a number of assertions to be made (depending on what pre-conditions are satisfied) about various *state observation* predicates after a *desired state* predicate is inferred.

(3) *Action* predicates :- Corresponding to actions to be taken in the plan, we have action predicates. Predicates corresponding to primitive actions may be designated as *time-setters* . In the rules

(grip) <- (close-gripper)

(ungrip) <- (open-gripper)

(posn-hand ?x ?y ?z) <- (move-robot-arm ?x ?y ?z)

close-gripper , *open-gripper* , *move-robot-arm* are examples of such predicates .

In the context of planning, a task may be decomposed into several subtasks. It is the structure of these subtasks which determine the plan necessary for executing the task. At the bottom of the hierarchy are found primitive tasks whose execution requires no planning to be carried out. The primitive tasks cannot be broken down any further. Which actions count as primitive depends on the available hardware and how

the Blocks World, the three basic robot arm operations considered as primitive are *open-gripper* , *close-gripper* and *move-robot-arm* The planner is started with a task, which it must reduce to these primitive actions. These primitive actions are also designated as *time-setters* The inferring of the *time-setters* causes an incrementing of the clock and moving ahead with respect to time. Through this mechanism a linear ordering of the states of the Blocks World with respect to time takes place.

3.2.3 Notion of Functional Dependency

The temporal predicates define relations between certain entities in the planning domain to certain attributes. In the domain of the Blocks World, the entities are the blocks about which attributes such as positional coordinates (x , y , z) are asserted at different points of time.

In general, the relations defined by the temporal predicates are functional in nature [9]. For example, consider the predicate *posn* with four arguements : *(posn ?b ?x ?y ?z)* . It describes the relation that the block *?b* is at the position whose coordinates are given by *?x* , *?y* , *?z* . We say that the last three arguments (positional coordinates) are functions of the first argument (block). A block uniquely determines the position in which it is situated at any given instant. Two different blocks cannot

be at the same position at any given instant. Formally, a functional relation F is said to exist among the arguments of a predicate P if a set of arguments at positions I of P uniquely determines a set of arguments at positions D of P.

F : I --> D for predicate P

sets I and D are known as set of independent (key) and dependent (non-key) arguments of P respectively. In the temporal predicates of the Blocks World, which relate entities or blocks to different attributes, the entities may be looked upon as the keys of the relation.

3.3 Semantics of Temporal Logic Programming Rules

In the present system, the plan-generation rules of the user have no explicit references to state and time. In this section, we outline a method by which these rules may be transformed to a set of rules where the state variables and their ordering become explicit in the formulation of the rules. Thus the method helps in clearly defining the temporal semantics of the user's rules.

To illustrate the above, let us consider the example of grasping a block in the Blocks World. The user defined rules are as follows

```
(grasp ?block) <- (is-grasp ?block)

(grasp ?block) <- (block ?block)
                  (clear ?block)
```

(handempty)

(is-posn ?block ?x ?y ?z)

(posn-hand ?x ?y ?z)

(grip)

The corresponding rules with explicit use of state variables as arguments of predicates are :

(grasp ?block ?S ?S) <- (is-grasp ?block ?S)

{ the desired state (?S) remains the same as the initial state (?S) when the state observation is satisfied at the initial state (?S) }

(grasp ?block ?Si ?Sf) <- (block ?block)
(clear ?block ?Si ?S1)
(handempty ?S1 ?S2)
(is-posn ?block ?x ?y ?z ?S2)
(posn-hand ?x ?y ?z ?S2 ?S3)
(grip ?S3 ?S4)
(= ?Sf (do-grasp ?block ?S4))

Here, every *desired state* predicate - *grasp, clear, handempty, posn-hand, grip* have two additional state variables as arguments. The desired state, on being achieved, transforms the Blocks-World from the first state to the second. Thus in our example,

?Si -----(grasp ?block)----> ?Sf

The *state observation* predicates, on the other hand, have only a single additional state variable in the list of arguments. They check for satisfaction in that particular state. In the example, *(is-posn ?block ?x ?y ?z ?S2)* is true when a block *?block* has positional coordinates *?x* , *?y* , *?z* in the state *?S2* .

The ordering of states, which was implicit in the original set of rules now becomes explicit. There is a sequence on the literals in the body of the rules. On examining the antecedents of the rules for the *desired state* predicates, we find that the planning world moves through a number of intermediate states in the process of transition from the initial state to the final state. In the example, *(grasp ?block)* causes the Blocks-World to transit from the initial state *?Si* to final state *?Sf*. In the process it moves through the intermediate states *?S1, ?S2, ?S3* and *?S4* if required.

By this mechanism, we can also reflect the linear ordering of states with respect to time. In general, the state that results on activating a *time-setter* follows the state that existed before activating the *time-setter* in time. The concept can be illustrated by considering the rules for *grip* .

```
(grip ?S ?S) (- (is-gripped ?S)

(grip ?Si ?Sf) (- (close-gripper ?Si ?Sf)
                  (= ?Sf (do-close-gripper ?Si))
```

where *(do-close-gripper ?s)* produces a state which immediately follows state *?s* in time. The incrementing in time is brought about by inferring the *time-setter - close-gripper* .

The plan for achieving the goal state is obtained as the unification of the second state variable in the *desired state* predicate of the goal. Looking at an example of the Blocks-World, let the initial configuration be :

```
(defasrt 1 (block A))

(defasrt 2 (block B))

(defasrt 3 (is-posn A 1 1 1 Si))

(defasrt 4 (is-posn B 2 2 2 Si))

(defasrt 5 (Ht A 2))

(defasrt 6 (Ht B 2))

(defasrt 7 (is-clear A  Si))

(defasrt 8 (is-clear B Si))

(defasrt 9 (is-posn-hand 3 3 3 Si))
```

Using the transformed rules where the state variables are explicit arguments of predicates, the goal *(on A B ?Si ?Sf)* produces the unification for *?Sf* as

(do-place A (do-open-gripper (do-posn A 2 2 4 (do-move-robot-arm 2 2 4 (do-grasp A (do-close-gripper (do-move-robot-arm 1 1 1 Si))))))) .

This is the plan for putting A on B. It is to be interpreted as - move the robot arm to (1 1 1), close the gripper thereby grasping A. After that, move the robot arm to (2 2 4) thereby positioning A to (2 2 4) and finally opening the gripper placing A at (2 2 4) which is on top of B.

In order to formulate the generalised scheme for transformation of the plan-generation rules without explicit state variables to a set of rules with the variables explicit, we partition the set of plan-generating rules in two classes -

(1) *desired state predication* rules : These rules typically have *desired state* predicates as consequents. Every *desired state* predicate, in general, has two rules associated with it. The first rule is the *observation* rule, in which it invokes a *state observation* predicate and *action* rule in which it invokes in general other *desired state* predicates, *time-setters* along with non-temporal and other *state observation* predicates.

(2) *state observation predication* rules : These rules typically have *state observation* predicates as

consequents. In the antecedents are other *state observation* predicates. This set of rules expresses relationships between different *state observation* predicates.

We define *elementary desired state* predicates as those which invoke *time-setters* or "primitive actions" in their *action* rules. That is these *desired state* predicates cannot be expressed in terms of any other *desired state* predicates. On the other hand, *compound desired state* predicates invoke other *desired state* predicates in their *action* rules. In the Blocks World, *on* is a *compound desired state* predicate, whereas, *grip* is an *elementary desired state* predicate.

The scheme for transforming *desired state predication* rules is as follows :

(1) If the rule is of the *observation* type then

(i) introduce two additional state variables as arguments of the *desired state* predicate in the consequent of the rule. These variables are identical. Let them be *?S* and *?S*.

(ii) introduce the same state variable *?S* as an additional argument of the *state observation* predicate in the antecedent of the rule.

(2) If the rule is of the *action* type then

(i) introduce two additional state variables as arguments of the *desired state* predicate in the

consequent of the rule. Let these be *?Si* and *?Sf* in that order respectively.

(ii) For the antecedents of the rule do

(a) If the antecedent involves non-temporal predication leave it as it is.

(b) If the antecedent is a *time-setter* then introduce *?Si* and *?Sf* as its additional arguments.

(c) If the antecedent involves *desired state* predication then add two additional intermediate state variables as arguments in the antecedent. If the antecedent is first in the list of antecedents of the rule then the first additional variable is *?Si*. If the antecedent is last in the list of antecedents of the rule then the second additional variable is *?Sint*.

(c) If the antecedent involves *state observation* predication then include the second state variable of the previous *desired state* predicate in the antecedent list as an additional argument of the *state observation* predicate.

(iii) If the *desired state* predicate (*dsp*) in the consequent of the rule is *elementary* (i.e. it invokes a *time-setter* (*ts*))
then add an antecedent of the form *(= ?Sf(ts ?Si))*
else (*dsp* is *compound*) add an antecedent of the

form (= ?Sf(#dsp arg_1 ... arg_n ?Sint))

where

dsp denotes *desired state* predicate

arg_1 ... arg_n denote arguments of dsp

?Sint denotes previous intermediate state i.e. second state variable argument of the last *desired state* predicate in the list of antecedents of the *desired state predication* rule.

The scheme for transforming the *state observation predication* rules is as follows :

(i) Introduce an additional state variable as argument of the *state observation* predicate in the consequent of the rule (?S).

(ii) For each of the antecedents of the rule do

(a) If the antecedent is non-temporal then leave it as it is

(b) If the antecedent involves *state observation* predicate then add the same variable (?S) as an additional argument of the predicate.

The above scheme of generating plan helps in writing of the *frame axioms* for the *independent state observation* predicates. We can recall the fact that a state variable may be unified to a constant (the *initial state* argument) or a list formed from successive applications of *desired state* predicates and *time-setters* on some initial state.

For every *independent state observation* predicate (*sopi*) with argument list (arg_1 ... arg_n) introduce the rules state that (i) the *independent state observation* predication is not affected in a state formed by the inferring of a *desired state* predicate which does not invoke it in the *observation* rule and (ii) the *independent state observation* predication is not affected by the inferring of a *desired state* predicate when the keys of the predications do not match. The rules are as follows :

(sopi arg_1 ... arg_n *?S) <-*
(not (initial-state ?s))
(is-time-setter (car ?s))
(sopi arg_1 ... arg_n *(setf (car ?S) (dsp-for-ts (car ?s))))*

(sopi arg_1 ... arg_n*) <-*
(not (initial-state ?S))
(not (sop-for-dsp sopi (car ?S)))
(sopi arg_1 ... arg_n *(predecessor ?S))*

(sopi arg_1 ... arg_n *?S) <-*
(not (initial-state ?s))
(= ?key$_1$ *(extr-key sopi* arg_1 ... arg_n*))*
(= ?key$_2$ *(extr-key (car ?S) (get-args (car ?S) ?S)))*
(not (equal ?key$_1$ *?key*$_2$*))*
(sopi arg_1 ... arg_n *(predecessor ?S))*

where

(initial-state ?s) : returns true if *?s* is bound to the initial state.

(is-time-setter ?x) : returns true if *?x* is bound to a *do-time-setter* else it returns nil.

(predecessor ?s) : returns the state preceeding state *?s* in time.

(extr-key pred arg_1 ... arg_n*)* : returns the key-list of the predicate *pred* from the list of arguments arg_1 ... arg_n

(sop-for-dsp ?sop ?dsp) : returns true if *?sop* is the *state observation* predicate for the *desired state* predicate *?dsp* else it returns nil.

(dsp-for-ts ?x) : returns the *elementary desired state* predicate which invoked the *time-setter - ?x.*

3.4 Need for maintaining History

AI databases are not static. Assertions come and go and the inferences they trigger must be kept upto date. A realistic database must keep track of what's true at various times, not just the present. As the planning world transits from one state to another, due to the occurrence of events, the various relations which are true in different states constitute the state descriptions at those instants. A record of what is true at different points of time constitutes the *history* of the planning world. In order to

do temporal analysis of generated plans, we cannot get away with representing just one situation (the "current" one) and must find some way to implement a database in which many different situations are stored. For example, in our Blocks World, the position of a block may change from one state to another (hence from time to time). In order to infer the position of the block at some earlier instant of time, a record of the varying positions of the block with time must maintained. Hence, we clearly feel the need of storing *history* .

In the context of logic programming, the temporal predicates need to refer to *history* for their satisfaction. These predicates constitute the method by which temporal question answering is possible. They are also referred to in the course of plan generation. In the course of plan generation, we may feel the need of placing block *A* at the position in which the block *B* was at a certain instant of time. Hence the plan generating rules may refer to the history of the Blocks World in order to infer where *B* was at the required instant of time. Only then can a suitable plan for placing *A* be generated.

3.5 Inference in Temporal Logic Programming

The execution of a *plan* causes the planning domain to move through several states. In the course of this transition, the time is incremented. Different state descriptions hold true at different instants of time. The

temporal logic programming system keeps track of the changing states and history of the world by means of temporal stacks. The procedure adopted for doing this will be explained in the following chapter. In addition, the system also has the ability to reason about time and comprehend time-related events. The system is able to answer questions regarding time in a typical logic programming fashion. It is able to answer questions regarding occurrence of events at particular time instants, as well as, examine the validity of properties over time intervals. The temporal logic used for this purpose is as follows :

There are two predicates :- *holds-inst* and *holds-int* . (*holds-inst ?p ?t*) is satisfied when a property *?p* is holds (i.e. true) at a time instant *?t* . (*holds-int ?p ?t*) is satisfied when a property *?p* holds over a time interval (list of time instants) *?t* .
To know, for example, when block *A* is on block *B* the user can set up a query : *(query (holds-inst (on A B) ?t))* or *(query (holds-int (on A B) ?t))* . If *(on A B)* is true at time instants Ø, 1, 2, 3, 7, 8, 9 - then the first query will return Ø, 1, 2 ... one after another. The second query, on the other hand, will return (Ø 3) and (7 9) as answers.

These two predicates can also be used in conjunction with other mutually exclusive primitive relations [1,8] that can hold between temporal intervals (or time instants) to make

meaningful queries.

There is a basic set of mutually exclusive primitive relations that can hold between temporal intervals. Each of these is represented by a predicate in the logic. These relationships are summarised in the definitions and the accompanying figure.

- *(During ?t1 ?t2)* : time interval *?t1* is fully contained within *?t2* .

- *(Starts ?t1 ?t2)* : time interval *?t1* shares the same beginning as *?t2* but ends before *?t2* ends.

- *(Finishes ?t1 ?t2)* : time interval *?t1* shares the same end as *?t2* but begins after *?t2* begins.

- *(Before ?t1 ?t2)* : time interval *?t1* is before interval *?t2* and they do not overlap in any way.

- *(Overlap ?t1 ?t2)* : time interval *?t1* starts before interval *?t2* and they overlap.

- *(Meets ?t1 ?t2)* : time interval *?t1* is before interval *?t2* but there is no interval between them i.e. *?t1* ends where *?t2* starts.

- *(Equal ?t1 ?t2)* : *?t1* and *?t2* are the same interval.

Relation	Pictorial Example
X before Y	XXX YYY

X equal	Y	XXX YYY
X meets	Y	XXXYYY
X overlaps	Y	XXX YYY
X during	Y	XXX YYYYYYY
X starts	Y	XXX YYYYY
X finishes	Y	XXX YYYYYY

Similar relationships can also be developed in conjunction with *holds-inst* for comparison of time points. For example, if we define

(Next ?rel1 ?rel2) :

(i) There exists time point *?t1* at which *?rel1* is true

(ii) There exists time point *?t2* at which *?rel2* is true

(iii) *?t2* = *?t1* + 1

then the rule for *Next* is given by :

```
(Next ?rel1 ?rel2) <- (holds-inst ?rel1 ?t1)
                      (holds-inst ?rel2 ?t2)
                      (eq ?t2 (+ ?t1 1))
```

By the development of such a temporal logic, we allow a powerful query language for the history. It allows us to reason over time instants as well as temporal intervals.

CHAPTER 4

IMPLEMENTATION OF A TEMPORAL LOGIC PROGRAMMING SYSTEM

4.1 Introduction

In this chapter we present an overview of the implementation of a temporal logic programming system. In this design we capture the formalism presented in the earlier chapter.

4.2 Notion of Temporal Stacks

Handling of state , time and history in the context of planning is achieved by suitably modifying the non-temporal logic programming shell. Along with the usual stacks required for the inference process (in an iterative fashion), the shell must now maintain what are called "temporal stacks" to keep track of the database changing with time. It is these stacks which store the time map of the plan as it is executed.

The predicates in the user's plan generating program can be of two types - *temporal* and *non-temporal*. The truth values of the temporal predicates change with time. The non-temporal predicates are unaffected by the state of the Blocks World and their truth values remain constant with

time. Thus *posn* may be regarded as a temporal predicate. *(posn A ?x ?y ?z)* will be satisfied by different values of *?x* , *?y* , *?z* as the position of block *A* changes with time. In contrast, *color* is independent of time in our Blocks World as there are no actions that cause a change in color. *(color A blue)* continues to be true in any state of the Blocks World. The truth values of the different temporal predicates describe the state of the world at any given time. So stacks have to be maintained for these predicates to reflect truly the notions of changing states, history and time. The top of the stack stores the "current" state of the planning world. We go down the stack to retrieve information about earlier instants and query about history.

4.3 Types of Temporal Stacks

By the nature of the arguments that the temporal predication involves, the stack required to be maintained for the temporal predicate varies accordingly. There are four distinct types of temporal stacks necessary (*type-1* , *type-2* , *type-3* , *type-4*) for the different classes of predicates. This classification helps in storing and retrieval of information about predicates.

(1) *type-1* stack :- This type of stack is maintained for temporal predicates which have no arguments at all. For example, *is-gripped* is a temporal predicate which has no arguments. It is either True or False at

different instants of time.

(2) *type-2* stack :- This type of stack is maintained for temporal predicates which have all their arguments as keys. *is-on* is a predicate of this kind. It defines a relationship between two entities (blocks). Both its arguments constitute the key and must be specified to uniquely define the relationship. For a given key it is either True or False with time.

(3) *type-3* stack :- This type of stack is maintained for predicates where none of the arguments are keys. *posn-hand* is an example of such predicate. The different coordinates of the *robot arm* are mentioned as attributes of the implicit entity - the *robot arm*.

Here, the attributes of a null key are recorded with time.

(4) *type-4* stack :- This type of stack is maintained for predicates where some of the arguments are keys. In the predicate *is-posn* we assert positional coordinates (attributes) about a key item - the block. The attributes of a key are recorded for keeping track of state, history and time.

4.4 Creation of Temporal Stacks

The temporal stacks are created at the beginning of execution of the logic program for plan generation. The list of predicates (relations) for which stacks are

to be maintained are decided upon examination of the plan generation rules supplied by the user. It is only for the independent *state observation* predicates that temporal stacks are maintained. Inference of the dependent state observations can be done from of the independent ones.

The type of the stacks to be maint deducible from the initial declarations of Therefore, on knowing the relations for which st to be maintained and the type of the stacl maintained the creation of stacks at run-time well-defined. Further, these stacks can be initi with the given facts in the initial configuration Blocks World.

For example, from the rules specified for Blocks World, we decide to maintain stacks for *is-* , *is-posn-hand* , *gripped* . The initialisation of stacks with the given data is as shown below. The initial description of the Blocks-World is given

```
(defasrt 1 (block A))
(defasrt 2 (block B))
(defasrt 3 (block C))
(defasrt 4 (block D))
(defasrt 5 (block E))
(defasrt 6 (block F))
```

```
(defasrt 7 (is-posn A 1 1 3))
(defasrt 8 (is-posn B 1 1 1))
(defasrt 9 (is-posn C 2 2 5))
(defasrt 10 (is-posn D 2 2 2))
(defasrt 11 (is-posn E 3 3 3))
(defasrt 12 (is-posn F 4 4 2))
(defasrt 13 (is-posn-hand 0 0 0))
```

The initialised stacks are as shown below :

```
(posn ((A) (0 T 1 1 3))
      ((B) (0 T 1 1 1))
      ((C) (0 T 2 2 5))
      ((D) (0 T 2 2 2))
      ((E) (0 T 3 3 3))
      ((F) (0 T 4 4 4)))

(posn-hand (() (0 T 0 0 0)))

(gripped (() (0 F)))
```

4.5 Update of Temporal Stacks

Regarding the update of the temporal stacks, we have to settle two major issues - (1) *when* to update the temporal stacks ? and (2) *How* to update the temporal stacks ?

4.5.1 When to update the temporal stacks ?

It may be worthwhile to recapitulate that the rules for a *desired state* predicate are written in a particular manner. The first rule has a *state observation* predicate in the antecedent. If it fails to detect the desired state (by the failure of the *state observation* predicate) then it causes the desired state to be brought into existence by activating appropriate *action* rule for it. The *state observation* predicates can be independent or dependent. Intuitively it is clear that to reflect the changes in state with time of the world, which is being modelled, updates of the temporal stacks must take place when

(1) *time-setter* is inferred implying that a primitive action has been performed.

(2) *desired state* predicate with an independent *state observation* predicate is inferred by using an action rule for it.

The justification of the previous statement is as follows. (1) When a *primitive action* or a *time-setter* is inferred, a state transition takes place and some of the existing predications will cease to become true and/or new predications may become true in the new state. This is because the *time-setter* was invoked only when the *desired state* predicate, of which it is the antecedent, could not be inferred by observation

alone. (2) When a *desired state* predicate is inferred by an action rule for it, a subtask for achieving the overall task , in the context of planning is accomplished. This accomplishment may have involved achieving other subsidiary desired states and invocation of *time-setters* in the process. The overall accomplishment has to be reflected in the truth value of the *state observation predicate* used by the *desired state* predicate as the observation predicate is independent. Let the state of the planning world before the inferring of the *desired state* predicate be S_i and the state after the inferring be S_f. The independent *state observation* predicate was not true in S_i but becomes true in S_f. Therefore, the temporal stack for it must be accordingly updated when the *desired state* predicate is inferred with the current time stamp. In this context, it may be worthwhile to mention that no update of the temporal stacks takes place when a *desired state* predicate with dependent *state observation* predicate is inferred. This is because the *state observation* predicate of the *desired state* predicate is in turn dependent on other independent *state observation* predicates, the stacks corresponding to which were properly updated when other *time-setters* were inferred in proving the first *desired state* predicate.

4.5.2 How to update the temporal stacks ?

The specification of the types of the temporal stacks provides significant clues regarding the procedures for updating the temporal stacks. In general, a temporal stack has assertions about attributes of key-items at different instants of time as its individual entries. The generalised representation of any temporal stack is as follows :-

(((key$_1$) (time$_1$ T/F attr$_1$ attr$_2$ ... attr$_n$)
(time$_2$ T/F attr$_1$ attr$_2$ ... attr$_n$)....
(time$_k$ T/F attr$_1$ attr$_2$ ... attr$_n$))

((key$_m$) (time$_x$ T/F attr$_1$ attr$_2$ ... attr$_k$)
(time$_y$ T/F attr$_1$ attr$_2$ ... attr$_k$)....
(time$_z$ T/F attr$_1$ attr$_2$ ... attr$_k$)))

where

(1) key$_1$, key$_2$, ... key$_i$, i = 1,2,... represent collection of entities needed for uniquely identifying the temporal relation. It may be nil in case of *type-3* stacks.

(2) time$_1$, time$_2$, ... time$_i$, i= 1,2,... represent different instants of time when the attributes of the key-items were asserted or retracted.

(3) T/F is a flag (either T or F) for denoting whether the relation involving the key-items and the attributes is True or False at the specified instant of time .

(4) $attr_1$, $attr_2$, ... $attr_i$, i= 1,2,... denote the different attributes which are asserted or retracted about the key-items at the specified instants of time. When the stack is of *type-2* there are no attributes in a stack entry. The key comprises of all the arguments .

The update of the temporal stacks involves two principal actions - assertion and retraction. The actions are comparable to the specification of "addlist" and "deletelist" in other planners such as STRIPS [4,1Ø]. An "addlist" is a list of facts that become true when an event becomes true ; a "deletelist" is a list of facts which cease to be true when the event occurs. Similarly in assertion we claim to make a certain relation involving key-items and its attributes true at a certain point of time. Retraction is just the opposite of assertion. Here, we claim that a certain relation involving key-items and its attributes ceases to be true when an event occurs. For the purpose of assertion and retraction, a special predicate *asserth* is provided in the temporal logic programming system. This can be used for storing and removing temporal facts with appropriate

time stamps. For example,

(asserth (is-posn A 1 2 3))

asserts the position of the block *A* to be *(1 2 3)* at the current instant of time in the position stack. Retraction of a relation can be achieved by asserting the negation of the relation. For example,

(asserth (not (is-clear A)))

retracts the fact that *A* is clear from from the database i.e. *A* ceases to be clear from the current time instant.

The set of rules, supplied by the user, is pre-processed and the temporal assertions to be made are included in the form of *(asserth ⟨relation⟩)* , where *⟨relation⟩* is any predication (or negation of predicate) of the observation kind, as antecedents of different *desired state* predicates. These predicates have *time-setters* as antecedents in the last rule for them. For example, the *desired state* predicate *posn-hand* has a set of rules given by

(1) *(posn-hand ?x ?y ?z) ⟨-*
(is-posn-hand ?x ?y ?z)

(2) *(posn-hand ?x ?y ?z) ⟨-*
(block ?block)
(is-grasp ?block)
((asserth (is-posn ?block ?x ?y ?z))

(@)

(fail)

(3) *(posn-hand ?x ?y ?z) <-*

(asserth (is-posn-hand ?x ?y ?z))

(@)

(fail)

(4) *(posn-hand ?x ?y ?z) <-*

(move-robot-arm ?x ?y ?z)

Here, the first rule for positioning the *robot arm* to a set of arbitrary coordinates checks whether the hand is in that position or not. If not, then the second rule fires. In this rule we first check for the pre-condition *(is-grasp ?block)* i.e. whether a block is grasped by the *robot arm* at the current instant. If it is, then the new position of the block is asserted to be the new position of the *robot arm* at the next time instant obtained by physically moving the *robot arm* .

The special predicate *(@)* has no arguments. As a goal this succeeds immediately. However, it has side-effects which alter the way backtracking works afterwards. The effect is to make inaccessible the place-markers for certain goals so that they cannot be satisfied. Hence, if *(@)* appears in some rule and is

satisfied, then the program becomes committed to *all choices made after the parent goal was invoked*. Therefore, when the second rule fails due to the last antecedent - *(fail)* , *(@)* prevents attempts to resatisfy *(is-grasp ?block)* or *(asserth (is-posn ?block ?x ?y ?z))* . It instead tries the third rule and so on till it is finally satisfied by the last rule. The effect of *asserth* is to place all its arguments (instantiated relations) temporarily into a list. Only when the *desired state* predicate gets satisfied by the rule with the *time-setter* in the antecedent, is the time incremented and the updates made permanent on the temporal stacks with the new time stamp.

Let us introduce two new terms - *elementary desired state* predicates and *compound desired state* predicates. The *elementary desired state* predicates are those which do not invoke any other *desired state* predicate, in order to be inferred. They cause a *time-setter* to be invoked when the observation rule fails. The *compound desired state* predicates are those which may invoke other *desired state* predicates in order to be inferred by their action rules. In the formulation of the Blocks World, predicates such as *posn* , *grasp* , *clear* , *on* etc. are examples of *compound desired state* predicates. They invoke *posn-hand* , *grip* , *ungrip* in order to be inferred by their action rules. Predicates such as *grip* , *ungrip* , *posn-hand* are examples of

elementary desired state predicates as they first observe and then invoke *time-setters* such as *close-gripper , open-gripper , move-robot-arm* in order to be inferred.

The method of pre-processing the plan-generation rules in order to generate the appropriate *asserth's* in the antecedents of the rules for the various *desired state* predicates is as follows :

(1) For all *elementary desired state* predicates (edsp) with *independent state observation* predicates (sopi), introduce the rule

```
(edsp) <- (asserth (sopi))
          (@)
          (fail)
```

In the Blocks World, *grip* is an *elementary desired state* predicate with the independent *state observation* predicate - *is-gripped* . So after pre-processing, the rules for *grip* become

```
(grip) <- (is-gripped)

(grip) <- (asserth (is-gripped))
          (@)
          (fail)

(grip) <- (close-gripper)
```

(2) For all *compound desired state* predicates (cdsp) with *independent state observation* predicates (sopi), examine the antecedents in the action rule for the *compound desired state* predicate. Assert the *independent state observation* predicate as an antecedent of the last *elementary desired state* predicate invoked in inferring the *compound desired state* predicate by the action rule. Include the *state observation* predicates of the other *desired state* predicates invoked before invoking the last *elementary desired state* predicate as pre-conditionss of the assert. For example, *posn* is a *compound desired state* predicate with an *independent statte observation* predicate *is-posn* . The action rule for *posn* is given by

```
(posn ?block ?x ?y ?z) <- (block ?block)
                          (grasp ?block)
                          (posn-hand ?x ?y ?z)
```

Here, *posn-hand* is the last *elementary desired state* predicate invoked in inferring *posn* by the action rule. Therefore the rules for *posn-hand* after pre-processing become as mentioned earlier in the section. The pre-processing of plan-generation rules helps in associating all update actions of temporal stacks with the inferring of *time-setters* .

4.6 Resetting Time

Time has to be reset and effects of temporal actions undone when the logic programming system enters the *backtracking* mode. In this context, handling of the temporal stacks while the logic programming system backtracks becomes important. Each *time-setter* when inferred is pushed into the predicate stack with a special flag. The *time stamp* generated after inferring the *time-setter* is also noted. As stated earlier, the rules for plan generation are pre-processed so that the temporal actions of *desired state* predicates , which are not *time-setters* are reflected as the update action of the terminal *time-setter* which will be invoked when trying to satisfy the *desired state* predicate. Therefore, while backtracking and undoing temporal assertions, it suffices to consider *time-setters* only. When a *time-setter* is popped out of the predicate stack, the various entries in the different temporal stacks with the *time stamp* of the *time-setter* are retracted.The clock is also decremented by one unit. This sets the state of the world to the one which existed before the backtracked *time-setter* was inferred.

For example, let the position of block *A* be (1 2 3) at the 7th instant of time. Owing to *(move-robot-arm 4 5 6)* at the 10th instant the position of *A* changes and is asserted to be (4 5 6). Therefore the position stack at the 10th instant is :

(posn ((A) (Ø T ...) . . . (7 T 1 2 3) (1Ø T 4 5 6)) . . .)

Now, in the backtracking mode, when *(move-robot-arm 4 5 6)* with the time-stamp of 1Ø is retracted, the assertion actions at the 1Øth instant are undone and the position stack becomes :

(posn ((A) (Ø T ...) . . . (7 T 1 2 3)) . . .) .

CHAPTER 5

ANALYSIS AND EVALUATION OF THE SYSTEM

5.1 Special Features Of The System

The temporal logic programming system is different from some of the existing systems in certain ways. The existing plan generation and robot problem-solving systems such as STRIPS operate as a forward deduction system. Forward rules (F - rules) are fired in the course of generation of the plan and change one state description into another. In this way , the sequence of actions needed for achieving the goal state is obtained. For modelling the changes in the state of the world being modelled, each forward rule must specify three components - precondition formula, delete list and add list. We have already encountered the last two terms. The precondition formula is a predicate calculus expression that must logically follow from the facts in a state description in order for the F-rule to be applicable to that state description.

The temporal logic programming system and the method outlined for prescribing rules for plan generation enable similar problem solving in a backward chained fashion. This allows construction of robot plans in an efficient fashion as we work backward from a goal expression to an initial

state description , rather than vice versa. The strategy employed for this purpose is to apply a conjunction of antecedents consisting of preconditions and assertions in the rules of the *desired state* predicates. This has been illustrated before. Along with its functioning as a backward chained system, the designed system performs the role of a *Time Map Manager (TMM)* [2]. It is able to store, update and retrieve from a long-term store of time tokens. The temporal understanding capability of the system is a consequence of its ability to handle time. It keeps track of the history of the world as a plan executes and is subsequently able to answer about validity of different relations at different instants of time in the course of execution of the plan.

5.2 Declarations required in Temporal Logic Programming

(1) (*Time-setters* $pact_1$ $pact_2$ ... $pact_n$)

(2) (*State-preds* (dsp_1 sop_1 n_1 (k_1 ... k_n))
(dsp_2 sop_2 n_2 (k_1 ... k_n'))

(dsp_k sop_k n_k (k_1 ... k_n'')))

where $pact_i$ denotes ith primitive action
dsp_i denotes ith desired state predicate
sop_i denotes ith state observation predicate

n_i denotes no. of arguments in ith desired state predicate

$(k_1 \ldots k_n)$ denotes the position of the key-items in the list of arguments of the desired state predicate.

These declarations are to be made by the user along with the set of rules for plan generation.

For example, in the formulation of the Blocks World, the declarations to be made by the user are :

```
(Time-setters open-gripper close-gripper move-robot-arm)

(State-preds (place is-place 4 (1))
             (posn is-posn 4 (1))
             (grasp is-grasp 1 (1))
             (clear is-clear 1 (1))
             (on is-on 2 (1 2))
             (handempty is-handempty ∅ ())
             (posn-hand is-posn-hand 3 ())
             (grip is-gripped ∅ ())
             (ungrip is-not-gripped ∅ ())
)
```

5.3 Representing rules in Temporal Logic Programming :

In general, the set of rules that will be used for plan generation and can be partitioned into three classes - (1) rules specifying the inferring of *desired state* predicates (2) rules specifying relationship between *state*

observation predicates , and (3) rules specifying inferring of *action* predicates.

Writing rules of class (1) is fairly straightforward. The first rule observes whether the desired state exists by invoking the corresponding *state observation* predicate. The subsequent rules specify the antecedent states needed to be achieved for inferring the overall desired state. Here, the user is expected to clearly write any temporal assertions which are indirect and do not follow directly from the given rules nor declarations about *desired state* and *state observation* predicates. To illustrate the point, we consider the rules for the *desired state* predicate *ungrip* . The rules are as follows

(1) *(ungrip) <- (is-not-gripped)*

(2) *(ungrip) <- (asserth (not (is-gripped)))*
(@)
(fail)

(3) *(ungrip) <- (open-gripper)*

The *desired state* predicate *ungrip* uses a *state observation* predicate which is not independent. The rule to infer it is given by

(is-not-gripped) <- (not (is-gripped))

Initially if the *robot arm* is gripped, then on achieving the desired state *ungrip* we need to assert that it is no

longer gripped. So we need to assert about *is-not-gripped* after inferring *ungrip*. This, in effect, implies that the independent *state observation* predicate *is-gripped* must be retracted after the inferring of *ungrip*. The antecedent *(asserth (not (is-gripped)))* achieves this. The second rule of *ungrip* needs to be explicitly specified by the user of the system.

Defining the relationships between the *state observation* predicates and defining rules for *action* predicates are illustrated below.

Some of the rules specifying relationships between *state observation* predicates are :

```
(is-place ?block ?x ?y ?z) <- (is-posn ?block ?x ?y ?z)
                              (not (is-grasp ?block))

(is-grasp ?block) <- (is-posn ?block ?x ?y ?z)
                     (is-posn-hand ?x ?y ?z)
                     (is-gripped)
```

The *action* rules are typically computational in nature which return true.

5.4 Scope for further improvement

In the course of developing an inteligent Logic Programming framework which would efficiently handle time, different problems had to be solved. Some of these offered direct solutions while some required more complicated

methods for solving.

One such problem which did not offer a direct solution was in connection with update of temporal stacks. As stated earlier, the temporal stacks are to be updated when a *time-setter* is inferred. When the *desired state* predicate of which the *time-setter* is the antecedent has an independent *state observation* predicate (with a temporal stack reserved for it), the update action is fairly staraightforward. After the inferring of the *time-setter* the *state observation* predicate has to be asserted to be true. However, when the *desired state* predicate in question uses a dependent *state observation* predicate, the update actions are not so obvious. Consider the following example of a restricted version of the Blocks World domain where the only *desired state* predicate is *on* . The plan generation rules and the user declarations are as follows :

```
(on ?block1 ?block2) <- (is-on ?block1 ?block2)

(on ?block1 ?block2) <- (do-stack ?block1 ?block2)

(is-on ?block1 ?block2) <- (is-posn ?block1 ?x1 ?y1 ?z1)
                           (is-posn ?block2 ?x1 ?y1 ?z-comp)
                           (height ?block1 ?h1)
                           (height ?block2 ?h2)
                           (= ?z-comp (add ?z1
                                          (product 0.5 ?h1)
                                          (product      0.5
```

```
?h2)))

                                (> ?z1 0)
                                (> ?z2 0)

(time-setters do-stack)

(state-preds (on is-on 2 (1 2))
             (() is-posn 4 (1))
```

Since the *state observation* predicate *is-on* is dependent on the independent observation *is-posn* , we do not maintain a separate srack for it. It is only the *is-posn* for which a temporal stack is maintained. Now, assume the initial declarations are as follows :

```
(block A)
(block B)
(height A 2)
(height B 2)
(is-posn A 1 1 1)
(is-posn B 2 2 2)
```

The desired state *(on A B)* has to be achieved by the action rule for *on* . So, the *time-setter -- do-stack* has to be inferred. In oreder to satisfy the observation *(is-on A B)* at this point of time (in the new state of the

Blocks World), the positional coordinates of *A* have to be changed. That is, after achieving the desired state *(is-posn A 2 2 4)* has to be asserted with the current time-stamp (1) in the position stack. *(is-posn B 2 2 2)* has to remain unaffected.

To impart intelligence to the system, so that it is capable of doing such deductions would involve a procedure of trial and error. We would have to satisfy a dependent *state observation* predicate by satisfying the independent observations on which it is dependent. This involves rearranging and asserting about the independent *state observation* predicates which appear as antecedents of different rules.

The correctness of such a proof method could not be clearly outlined. Hence, at this stage the system would expect an explicit declaration from the user of the form :

```
(on ?block1 ?block2) (- (is-on ?block1 ?block2)

(on ?block1 ?block2) (- (is-posn ?block2 ?x ?y ?z)
                        (height ?block1 ?h1)
                        (height ?block2 ?h2)
                        (= ?z-comp (add ?z
                                        (product 0.5 ?h1)
                                        (product 0.5
?h2)))
                        (asserth (is-posn ?block1 ?x ?y
?z-comp))
```

```
(@)
(fail)

(on ?block1 ?block2) <- (time-setter)
```

5.5 Conclusion

The major achievement of the Temporal Logic Programming formalism presented in this thesis is that it allows Planning and temporal Understanding to take place in a logic programming framework. It is becoming possible to handle temporal notions in the logic programming environment.

Further, the formalism circumvents the problem of enumeration of frame axioms. The necessity of maintaining explicit state variables as additional arguments of predicates in the rules of the logic program and stating the effects and non-effects of actions is obviated in this approach. The formalism permits a strong basis for reasoning with time. The query mechanism for deduction of temporal relationships is powerful. The handling of time. state and history in this formalism adds a new dimension to planning techniques. It is now possible to generate planning actions with reference to past situations. To illustrate in terms of our familiar Blocks World, it is possible to plan "Place block A where block B was when block C was on top of block D" using the temporal logic programming formalism.

REFERENCES

(1) Allen,J.F 1984 : Towards a General Theory of Action Time, Artificial Intelligence, 23 pp 123-154

(2) Charniak,E and McDermott,D 1985 : Introduction Artificial Intelligence,Addison-Wesley Publishing Compan

(3) Clocksin,W.F and Mellish,C.S 1986 : Programming Prolog, Springer-Verlag, Berlin

(4) Fikes,R.E and Nilsson,N.J 1971 : STRIPS - a new appr to the application of theorem proving to problem solv Artificial Intelligence, 2 (314) pp 189-208

(5) Green,C 1969 : Theorem proving by resolution as a b for question answering systems, In MI4, pp 183-205

(6) Kowalski,R 1974 : Logic for Problem Solving, Memo no Dept. of Computational logic, Univ. of Edinburgh

(7) Kowalski,R 1979 : Logic for Problem Solving, New Yor North Holland

(8) Moszkowski,B 1986 : Executing temporal logic progr Cambridge University Press

(9) Nakamura,K 1986 : Control of Logic Program Execu based on Functional Relation, 3rd International Conferenc on Logic Programming.

APPENDIX-1
==========

```
(comment "the following are the set of plan generation rules
          to be used for the Blocks-World model ")


(comment "************************************")
(comment " rules for desired state predicates ")
(comment "************************************")

(defasrt pl1 (place ?block ?x ?y ?z) <-
             (is-place ?block ?x ?y ?z)
)
(defasrt pl2 (place ?block ?x ?y ?z) <-
             (block ?block)
             (posn ?block ?x ?y ?z)
             (ungrip)
)


(defasrt po1 (posn ?block ?x ?y ?z) <-
             (is-posn ?block ?x ?y ?z)
)
(defasrt po2 (posn ?block ?x ?y ?z) <-
             (block ?block)
             (grasp ?block)
             (posn-hand ?x ?y ?z)
)


(defasrt gr1 (grasp ?block) <-
             (is-grasp ?block)
)
(defasrt gr2 (grasp ?block) <-
             (block ?block)
             (clear ?block)
             (handempty)
             (is-posn ?block ?x ?y ?z)
             (posn-hand ?x ?y ?z)
             (grip)
)


(defasrt cl1 (clear ?block) <-
             (is-clear ?block)
)
(defasrt cl2 (clear ?block) <-
             (block ?block1)
             (block ?block)
             (is-on ?block1 ?block)
             (grasp ?block1)
             (handempty)
)
```

```
(defasrt on1 (on ?block1 ?block) <-
             (is-on ?block1 ?block)
)
(defasrt on2 (on ?block1 ?block) <-
             (block ?block1)
             (block ?block)
             (clear ?block)
             (clear ?block1)
             (is-posn ?block ?x ?y ?z)
             (ht ?block ?h)
             (ht ?block1 ?h1)
             (= ?z-comp (calc-ht ?z ?h ?h1))
             (place ?block1 ?x ?y ?z-comp)
)


(defasrt he1 (handempty) <-
             (is-handempty)
)
(defasrt he2 (handempty) <-
             (= ?empx (find-empty-posn-x))
             (= ?empy (find-empty-posn-y))
             (= ?empz (find-empty-posn-z))
             (posn-hand ?empx ?empy ?empz)
             (ungrip)
)


(defasrt ph1 (posn-hand ?x ?y ?z) <-
             (is-posn-hand ?x ?y ?z)
)
(defasrt ph2 (posn-hand ?x ?y ?z) <-
             (move-robot-arm ?x ?y ?z)
)


(defasrt gr1 (grip) <-
             (is-gripped)
)
(defasrt gr2 (grip) <-
             (close-gripper)
)


(defasrt ug1 (ungrip) <-
             (is-not-gripped)
)
(defasrt ug2 (ungrip) <-
             (asserth (not (is-gripped)))
             (@)
             (fail)
)
(defasrt ug3 (ungrip) <-
             (open-gripper)
)
```

```
(comment "*****************************************************************")
(comment " rules expressing relations among state observation predicates ")
(comment "*****************************************************************")

(defasrt ipl (is-place ?block ?x ?y ?z) <-
             (is-posn ?block ?x ?y ?z)
             (not (is-grasp ?block))
)


(defasrt igr (is-grasp ?block) <-
             (is-posn ?block ?x ?y ?z)
             (is-posn-hand ?x ?y ?z)
             (is-gripped)
)


(defasrt icl (is-clear ?block) <-
             (block ?block)
             (not (is-loaded ?block))
)


(defasrt isl (is-loaded ?block) <-
             (block ?block1)
             (block ?block)
             (is-on ?block1 ?block)
)


(defasrt ion (is-on ?block1 ?block) <-
             (block ?block1)
             (block ?block)
             (not (eq ?block1 ?block))
             (is-posn ?block ?x ?y ?z)
             (ht ?block ?h)
             (ht ?block1 ?h1)
             (= ?z-comp (calc-ht ?z ?h ?h1))
             (is-posn ?block1 ?x ?y ?z-comp)
)


(defasrt ihe (is-handempty) <-
             (not (is-gripped))
)


(defasrt ing (is-not-gripped) <-
             (not (is-gripped))
)
```

```
(comment "*************************************")
(comment "rules defining temporal relationships")
(comment "*************************************")


(defasrt dur (during ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (> (car ?t1) (car ?t2))
             (< (cadr ?t1) (cadr ?t2))
)

(defasrt str (starts ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (eq (car ?t1) (car ?t2))
             (< (cadr ?t1) (cadr ?t2))
)

(defasrt fin (finishes ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (eq (cadr ?t1)(cadr ?t2))
             (>  (car ?t1) (car ?t2))
)

(defasrt bef (before ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (< (cadr ?t1) (car ?t2))
)

(defasrt ovr (overlaps ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (< (car ?t1) (car ?t2))
             (> (cadr ?t2) (cadr ?t1))
             (> (cadr ?t1) (car ?t2))
)

(defasrt met (meets ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (eq (add1 (cadr ?t1)) (car ?t2))
)

(defasrt eql (equal ?t1 ?t2) <-
             (not (null ?t1))
             (not (null ?t2))
             (eq (car ?t1) (car ?t2))
             (eq (cadr ?t1) (cadr ?t2))
)
```

```
(comment "********************")
(comment " user declarationss ")
(comment "********************")

(time-setters open-gripper close-gripper move-robot-arm)

(state-preds (place is-place 4 (1))
             (posn is-posn 4 (1))
             (grasp is-grasp 1 (1))
             (clear is-clear 1 (1))
             (on is-on 2 (1 2))
             (handempty is-handempty Ø ())
             (posn-hand is-posn-hand 3 ())
             (grip is-gripped Ø ())
             (ungrip is-not-gripped Ø ())
)

(comment "**********************************")
(comment "various computational definitions")
(comment "**********************************")


(defcomp close-gripper dum)
(defcomp move-robot-arm dum1)
(defcomp open-gripper dum)
(de dum () t)
(de dum1 (a b c) t)

(de calc-ht (a b c)
    (cond [(and (numberp a) (numberp b) (numberp c))
           (fix (add a (product .5 b) (product .5 c)))]
          [t (list 'a_d_d a (list 'p_r_o_d .5 b) (list 'p_r_o_d .5 c))]

(setq $countx 47)
(setq $county 47)
(setq $countz 47)
(de find-empty-posn-x ()
    (setq $countx (add1 $countx))
    (readlist (append (list 'e 'm 'p '- 'X) (list (ascii $countx))))
)
(de find-empty-posn-y ()
    (setq $county (add1 $county))
    (readlist (append (list 'e 'm 'p '- 'Y) (list (ascii $county))))
)
(de find-empty-posn-z ()
    (setq $countz (add1 $countz))
    (readlist (append (list 'e 'm 'p '- 'Z) (list (ascii $countz))))
)

(defcomp eq eq)
(defcomp > >)
(defcomp < <)
(defcomp and and)
(defcomp null null)
(defcomp eq eq)
(defcomp equal equal)

(comment "**********************************************************
```

APPENDIX-2

```
(comment "*************************************************************")

(comment "The following are the set of assertions which describe an
          initial state of the Blocks-World (refer APPENDIX - 3).
          There are six blocks - A,B,C,D,E,F and their heights and
          positions are as indicated by "Ht" and "is-posn" predicates.
          We find that A is on B, C is on D, E and F are clear.
          The hand is at the coordinates Ø,Ø,Ø as indicated by
          "is-posn-hand".
          The clock is set at Ø. ")

(comment "*************************************************************")

(defasrt 1 (block A))
(defasrt 2 (block B))
(defasrt 3 (block C))
(defasrt 4 (block D))
(defasrt 5 (block E))
(defasrt 6 (block F))
(defasrt 7 (ht A 2))
(defasrt 8 (ht B 2))
(defasrt 9 (ht C 2))
(defasrt 1Ø (ht D 4))
(defasrt 11 (ht E 6))
(defasrt 12 (ht F 4))
(defasrt 13 (is-posn A 1 1 3))
(defasrt 14 (is-posn B 1 1 1))
(defasrt 15 (is-posn C 2 2 5))
(defasrt 16 (is-posn D 2 2 2))
(defasrt 17 (is-posn E 3 3 3))
(defasrt 18 (is-posn F 4 4 2))
(defasrt 19 (is-posn-hand Ø Ø Ø))


(comment "*************************************************************
```

```
(comment "An interactive session with the
          Temporal Logic Programming system")
(comment "*************************************************")
(comment "Blocks-World-rules is the file in APPENDIX - 1")
(comment "Blocks-World-description is the file in APPENDIX - 2")
(comment "*************************************************")


[1](dskin Blocks-World-rules)

[load Blocks-World-rules]
(Blocks-World-rules)


[2](dskin Blocks-World-description)

[load Blocks-World-description]
(Blocks-World-description)


(comment "We generate a plan in the following way")

[3](plan (on C E))

The plan is as follows ...

 The CLOCK is set to 4
[Ø] (move-robot-arm 2 2 5) --> [1]
[1] (close-gripper) --> [2]
[2] (move-robot-arm 3 3 7) --> [3]
[3] (open-gripper) --> [4]
nil

(comment "The plan is in terms of time-setters
          which cause the clock to be incremented on being inferred")

(comment "To do temporal analysis of the Blocks-World we use the
          function 'query' ")


(comment "The following finds all the time points at which E is clear")

[4](query (holds-inst (clear E) ?t))

proved !
*** answer is :- ***
?t = Ø

*** more answers wanted ? (y/n) y
*** answer is :- ***
?t = 1

*** more answers wanted ? (y/n) y
*** answer is :- ***
?t = 2

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil
```

```
(comment "The following examines the time
          intervals in which E is clear")

[5](query (holds-int (clear E) ?t))

proved !
*** answer is :- ***
?t = (Ø 2)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil

(comment "The following query finds the positions of
          the different blocks at 4th time instant")

[6](query (holds-inst (posn ?block ?x ?y ?z) 4))

proved !
*** answer is :- ***
?block = A
?x = 1
?y = 1
?z = 3

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = B
?x = 1
?y = 1
?z = 1

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = C
?x = 3
?y = 3
?z = 7

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = D
?x = 2
?y = 2
?z = 2

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = E
?x = 3
?y = 3
?z = 3

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = F
?x = 4
?y = 4
?z = 2

*** more answers wanted ? (y/n) y
*** no more answers possible .
```

```
(comment "the following query finds the different
          positions of all the blocks in different
          time intervals")

[7](query (holds-int (posn ?block ?x ?y ?z) ?t))

proved !
*** answer is :- ***
?block = A
?x = 1
?y = 1
?z = 3
?t = (Ø 4)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = B
?x = 1
?y = 1
?z = 1
?t = (Ø 4)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = C
?x = 2
?y = 2
?z = 5
?t = (Ø 2)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = D
?x = 2
?y = 2
?z = 2
?t = (Ø 4)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = E
?x = 3
?y = 3
?z = 3
?t = (Ø 4)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = F
?x = 4
?y = 4
?z = 2
?t = (Ø 4)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = C
?x = 3
?y = 3
?z = 7
?t = (3 4)

*** more answers wanted ? (y/n) y
*** no more answers possible .
```

```
(comment "The following query tests for the
          'equality' of temporal intervals")

[8](query (holds-int (on A B) ?t1)
          (holds-int (clear F) ?t2)
          (equal ?t1 ?t2))

proved !
*** answer is :- ***
?t1 = (Ø 4)
?t2 = (Ø 4)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil


(comment "The following query tests whether two
          time-intervals meet or not")

[9](query (holds-int (on C D) ?t1)
          (holds-int (on C E) ?t2)
          (meets ?t1 ?t2))

proved !
*** answer is :- ***
?t1 = (Ø 2)
?t2 = (3 4)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil


(comment "The following query finds the time intervals in which
          the robot arm was gripping some block")

[1Ø](query (holds-int (grip) ?t1))

proved !
*** answer is :- ***
?t1 = (2 3)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil

(comment "Now, let us stop doing temporal analysis for a minute and do
          something more to the Blocks-World.
          Note when we plan again, we have the alternative of resetting
          the clock and starting all over from the initial state or
          continuing from the point where we left")
```

```
[11](plan (on D B) (clear E) (grasp F))

*** Continue from previous state (y/n) ? y
The plan is as follows ...

 The CLOCK is set to 18
[4] (move-robot-arm 1 1 3) --> [5]
[5] (close-gripper) --> [6]
[6] (move-robot-arm emp-XØ emp-YØ emp-ZØ) --> [7]
[7] (open-gripper) --> [8]
[8] (move-robot-arm 2 2 2) --> [9]
[9] (close-gripper) --> [1Ø]
[1Ø] (move-robot-arm 1 1 4) --> [11]
[11] (open-gripper) --> [12]
[12] (move-robot-arm 3 3 7) --> [13]
[13] (close-gripper) --> [14]
[14] (move-robot-arm emp-X1 emp-Y1 emp-Z1) --> [15]
[15] (open-gripper) --> [16]
[16] (move-robot-arm 4 4 2) --> [17]
[17] (close-gripper) --> [18]
nil

(comment "Now let us do temporal analysis")

[12](query (holds-int (clear E) ?t))

proved !
*** answer is :- ***
?t = (Ø 2)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?t = (15 18)

*** more answers wanted ? (y/n)
*** no more answers possible .

ok.
nil


(comment "The following query finds the positions of
          block A over all time-intervals")

[13](query (holds-int (posn A ?x ?y ?z) ?t))

proved !
*** answer is :- ***
?x = 1
?y = 1
?z = 3
?t = (Ø 6)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?x = emp-XØ
?y = emp-YØ
?z = emp-ZØ
?t = (7 18)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil
```

```
(comment "The following query finds all the
          intervals in which any
          block was loaded")

[14](query (holds-int (is-loaded ?block) ?t))

proved !
*** answer is :- ***
?block = B
?t = (Ø 6)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = B
?t = (11 18)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = D
?t = (Ø 2)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = E
?t = (3 14)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil


(comment "By the following query we test whether one time
          interval is wholly contained in another")

[15](query (holds-int (on C E) ?t1)
           (holds-int (clear F) ?t2)
           (during ?t1 ?t2))

proved !
*** answer is :- ***
?t1 = (3 14)
?t2 = (Ø 18)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil


(comment "The following query tests whether the
          intervals over which two relationships are
          true do overlap or not")

[16](query (holds-int (on A B) ?t1)
           (holds-int (clear D) ?t2)
           (overlaps  ?t1 ?t2))

proved !
*** answer is :- ***
?t1 = (Ø 6)
?t2 = (3 18)
```

```
[17](query (holds-int (posn C ?x ?y ?z) ?t))

proved !
*** answer is :- ***
?x = 2
?y = 2
?z = 5
?t = (Ø 2)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?x = 3
?y = 3
?z = 7
?t = (3 14)

*** more answers wanted ? (y/n) y
*** answer is :- ***
?x = emp-X1
?y = emp-Y1
?z = emp-Z1
?t = (15 18)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil

(comment "The following query finds the intervals in which D is clear"

[18](query (holds-int (clear D) ?t))

proved !
*** answer is :- ***
?t = (3 18)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil

(comment "The following query highlights
          an interesting facility.
          We can backtrack for the predicates
          'holds-int' and 'holds-inst' in our temporal
          logic programming system.
          In this case it is the second unification for ?t1 = (3 14)
          which answers the composite query")

[19](query (holds-int (posn C ?x ?y ?z) ?t1)
           (holds-int (clear D) ?t2)
           (starts ?t1 ?t2))

proved !
*** answer is :- ***
?x = 3
?y = 3
?z = 7
?t1 = (3 14)
?t2 = (3 18)

*** more answers wanted ? (y/n) y
*** no more answers possible .
```

```
(comment "The following illustrates when a
          time-interval is said to
          occur before another")

[20](query (holds-int (on C E) ?t1)
           (holds-int (grasp F) ?t2)
           (before ?t1 ?t2))

proved !
*** answer is :- ***
?t1 = (3 14)
?t2 = (18 18)

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil


(comment "The flexibility which the temporal system
          offers allows us to define new temporal relations
          as and when required.
          We illustrate this by defining 'always-true' and 'sometimes-true
          below.
          'current-instant' is a function which returns the current value
           of the clock")

[21](defasrt alt (always-true ?relation) <-
                 (holds-int ?relation ?t)
                 (eq (car ?t) 0)
                 (eq (cadr ?t) (current-instant)))

always-true

[22](defasrt sot (sometimes-true ?relation) <-
                 (holds-int ?relation ?t)
                 (not (null ?t)))

sometimes-true

[23](query (always-true (clear F)))

proved !
*** more answers wanted ? (y/n) n
ok.
nil


(comment "The following query finds all the blocks .
          which have not been moved at all during the
          state transition of the Blocks-World")
```

```
[24](query (always-true (posn ?block ?x ?y ?z)))

proved !
*** answer is :- ***
?block = B
?x = 1
?y = 1
?z = 1

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = E
?x = 3
?y = 3
?z = 3

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = F
?x = 4
?y = 4
?z = 2

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil


(comment "The following query finds all the blocks
          which were grasped at some point of time")

[25](query (sometimes-true (grasp ?block)))

proved !
*** answer is :- ***
?block = C

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = C

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = A

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = D

*** more answers wanted ? (y/n) y
*** answer is :- ***
?block = F

*** more answers wanted ? (y/n) y
*** no more answers possible .

ok.
nil
```

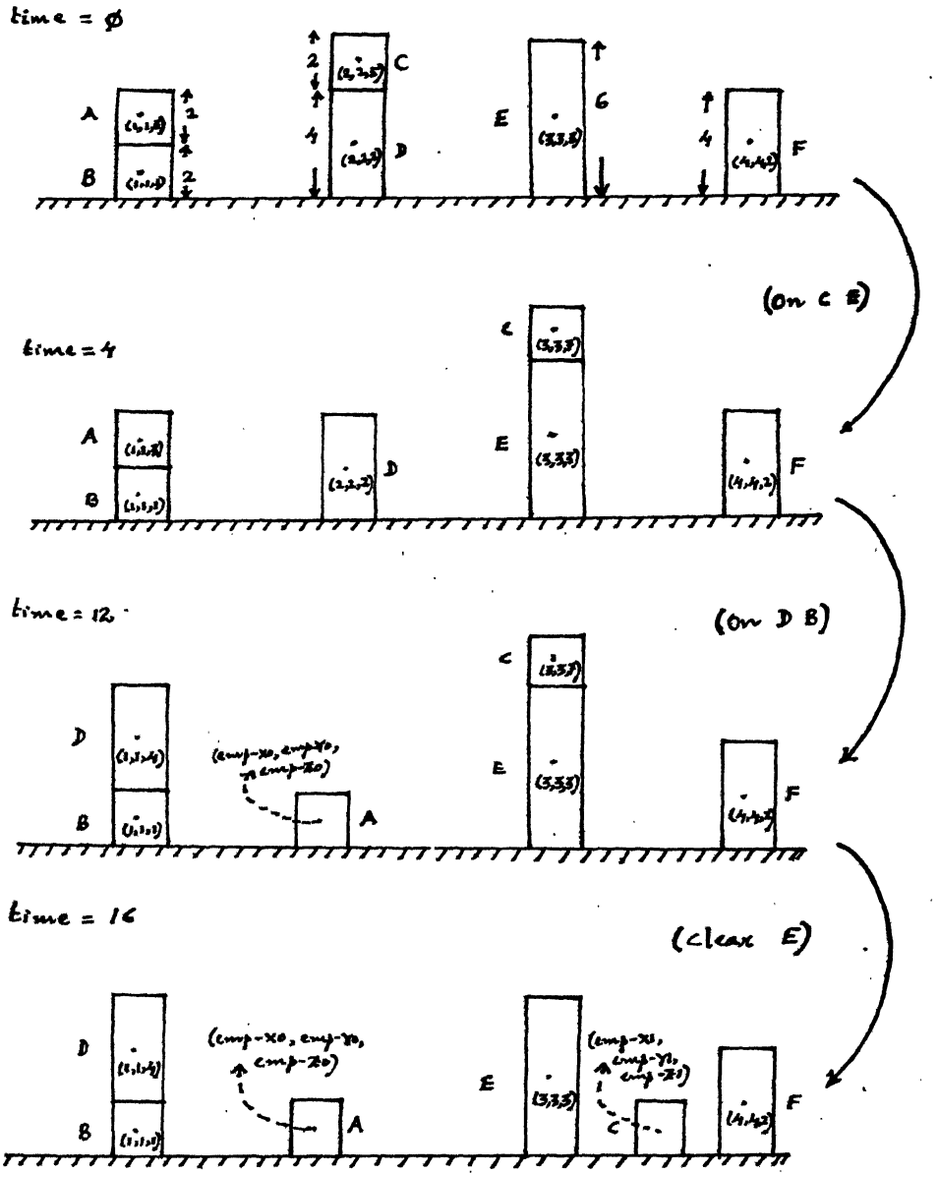


FIG. ILLUSTRATING CHANGING STATES OF BLOCKS-WORLD AS PLAN (APPENDIX-3) EXECUTES